TWO PLAYER BOARD GAME

Rohit Shindhe
190624325
Matthew Huntbach
MSc. Big Data Science

***Abstract* — For a significant amount of time, the minimax algorithm and AI developments were studied to achieve an ideal improvement in gaming fields, such as chess. Several generations are trying to update the pruning code and the evaluation feature effectiveness in these areas. There are robust algorithms in the field of gaming to handle various complex situations.**

When evaluating the winning chances of our version struggles with different searching algorithm depths against the online test, we find that our heuristics with a minimax algorithm are perfect in the early stages of zero-sum match. While some of the nodes in the game tree have no impact on the final result of minimax algorithm, we are using other method alpha-beta algorithm that has a great effect and efficiency to prune the number of irrelevant nodes.

In an adversarial search algorithm there is a concept Alpha-Beta pruning which uses pruning of trees to improve the minimax search of data. This method allows two opponents to get the best result while analyzing a search tree. The best examples of this approach are modern chess engines, which use Alpha-beta algorithm to calculate the moves. This paper explains the steps of the Alpha-beta pruning and how it is implemented in chess engines. We realize how a 64 year old Alpha-Beta algorithm is used in chess engine. (Kang, Wang and Hu, 2019)

Keywords— Alpha-Beta, Minimax, Chess Engine, Algorithm, Artificial intelligence, Search Tree, Pruning, Heuristic.

I. INTRODUCTION

Minimax algorithm in the gaming zone Chess, Backgammon, Tic-tac-toe and Connect-4 has already made progress. In addition as the generations continue to strengthen the search algorithms in AI, machine learning technologies, it has allowed superhuman intelligence to grow. In 1997 Deep Blue which is designed by IBM defeated world chess champion Garry Kasparov. Furthermore, since minimax has been studied continuously by experts through self-playing, a new cutting edge intelligent player emerged known as Alphazero which is designed and developed by Google's DeepMind Company.

In 1956 John McCarthy holds the credit for inventing the Alpha-Beta search algorithm. In 1957 the first program was released widely. In 1961 the first program that is capable for believable play is made using the Alpha-Beta Algorithm as its base. (D. E. Knuth, 1975)

The alpha-beta algorithm is one of the most common algorithm in searching games especially in adversary board games such as chess and checkers. It is more powerful than the simple brute force minimax algorithm because it makes it possible to prune a large number of nodes with the correct final value in the tree. With the increasing search depth the number of nodes visited by the algorithm increases exponentially. Chess game has an approximately 10^{120} potential moves. We can use search algorithms to automate and optimize the search process. (P.Winston, n.d.)

Alpha beta search algorithm allows computers to search all possible moves and finally find the best result.

II. RELATED WORK

The brute-force programs are considered to be most successful algorithms, as they mainly depend on program speed and optimization to find more positions and obtain more accurate evaluations. There are many other related AI programs which are listed in this paper. In AI attempts KnightCap is considered as best while playing with computer chess. It uses a variation called Temporal Difference learning machine algorithm also known as TD leaf (λ). In TD learning we can learn to adjust predictions and match more accuracy. (Baxter, n.d.)

By deep search we can obtain accuracy that are closer to the evaluation models. Through multiple games the feedback is passed to programs so that it can learn the evaluation function and adjust the weight values to fine tune the evaluation criteria. This can also be improved by machine learning.

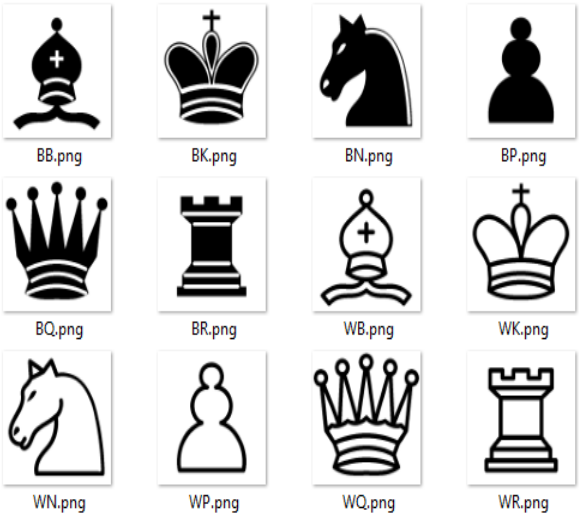
Once the process of search tree is optimized, then the most important part of the computer chess program would be the evaluation function. The results of optimization and brute-force alpha-beta search are compared, but with the forward pruning method it is even possible to produce small tree searches. To minimize the search tree is very important because the smaller the search tree the more information we get that can be added to evaluation function. (Bonet, n.d.)

A. Literature Review

Chess is one of the most well-known two player board game which is played on a board of 64 squares arranged in a grid of 8x8. (Russell and Norvig, n.d.)

When the game starts, each player arranges 16 of their chess coins on the board in a standardized order. These chess coins will be moved to attack, defend, threat, and capture the opponent's coins.

The aim of the game is to capture the opponent's "King" coin. The game will finish either when a king is captured or when the condition for stalemate/checkmate has met. The protocols of chess are published by FIDE, a World Chess Federation. (Chess, 2020)



Chess Coins Images

B. *Chess Coins and their Movements*

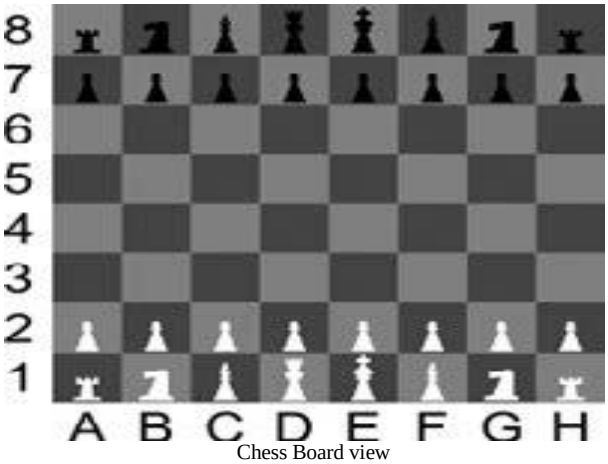
Each side of player would be given 16 coins, with 8 different roles consisting of 2 rooks, 2 knights, 2 bishops, a queen, a king and 8 pawns with each role has their own unique movement. The only thing in common is that the move path cannot be looped on the board. Which means that when a coin has reached end of the chess board, they must stop and they have to return all the way back if they want to reach another end.

Movement: Pawns can move forward to an empty square immediately next to it on the same path, or on its first move pawn can advance two squares along the same path, provided both squares are empty.

Pawn can capture an opponent's coin on a square diagonally in front of it on an adjacent path, by moving to that square. A pawn has two special moves: the en passant capture and promotion. Rook can move any number of squares along a rank or path, but cannot leap over other coins. Along with the king, a rook is involved during the king's castling move. Bishop can move diagonally in any range without leap over other coins. Knight's move are different than any other coins, they move in an "L" letter shape with total of 3 square move (i.e. two squares vertically and one square horizontally, or two squares horizontally and one square vertically). It is the only coin that can leap over other coins. Queen is the most powerful coin of all. Queen moves is the combination of bishop's and rook's move. King moves in any direction only for one square. Castling is a special move for the king that also involves shifting a rook. (Russell and Norvig, n.d.)

C. *Board Representation*

Computers cannot view at a chess board a similar way that humans do. Where we see the chess board the machine must be able to read and examine each of the coins as independent items. First we should place the board in a format that the machine can comprehend. This should be possible utilizing lists, arrays, or similar sets of data. It ought to be noticed that the manner in which a program stores and gets to this data can significantly affect its performance as millions of moves are being handled and by extension data-structures being accessed. The coins are put in one of two different ways, coin-centric and board-centric. Coin portrayal is done by storing the remaining coins (the ones not captured at this point of the game) on the board in the data structure. One technique for storing is with a bit board approach, with one 64-coin word for each coin type, with bits to relate their inhabitance, or board position. The other strategy, board-centric is done by looking at each square and figuring out what is there, vacant or occupied with what coin, and storing it along these path as coin centric. (Vuckovic, 2012)



Chess Board view

D. *Chess Engine View*

How to make a machine to play Chess? There are several options to do it.

One option is to use brute force computing power to calculate the entire game. But that it is highly unlike to use with the current technology.

The other option is to use limited number of moves and evaluate positions, that results to win. This option is known as Alpha-Beta algorithm that works best than the previous option. This algorithm is similar to minimax tree structure. (P.Winston, n.d.)

III. Methodology

A. *Minimax*

Minimax is a search algorithm. To restrict the potential loss to a possibility of worst case scenario, a decision rule is used in artificial intelligence, and decision theory. The minimax value of a player is the lowest value that the other players can compel the player to receive, without knowing the opponent's actions; similarly, it is the highest value that the player can be sure to get when they know the actions of the other players. Its formal definition is:

$$\overline{v_i} = \min_{a_{-i}} \max_{a_i} v_i(a_i, a_{-i})$$

v i _ = max a i min a − i v i (a i , a − i) {\display style {\underline {v_{i}}}=\max _{a_{i}}\min _{a_{-i}}{v_{i}(a_{i},a_{-i})}} (Kang, Wang and Hu, 2019)

Minimax is a bottom up algorithm. It starts to search from bottom and moves back to root node. For each position of the node a value is assigned with higher being better for the current player. The higher numbers the better position. (D. E. Knuth, 1975)

Minimax searches the whole tree to find the best move or it has to be set a certain depth. The algorithm cannot skip or cut any branches.

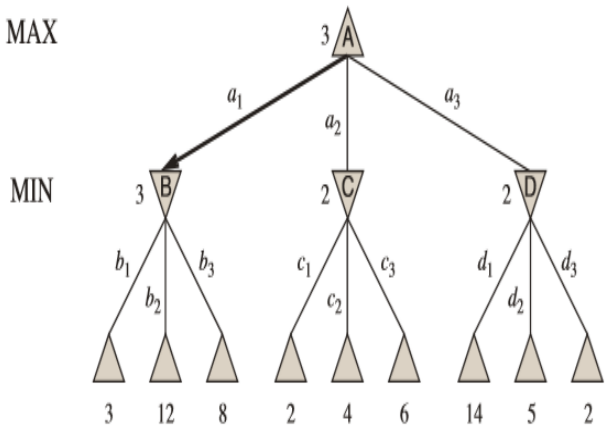


Figure. 1

The above figure 1, illustrates the process of the algorithm that the first node "A" starts down to the bottom left three nodes and uses the function called utility on them having the values 3, 12, and 8, respectively. At this point of level it chooses the minimum of these values 3, and returns it to the backed value of node B. The same process is carried and backed up values 2 for C and 2 for D. At the end, finally we choose the maximum value in 3, 2, 2, and 2 is the backed up value for the root node 3.

In the game tree, minimax algorithm executes a complete depth-first exploration. At each point there are b legal moves, for the maximum depth m in the tree, with the time complexity of the minimax algorithm is O(bm). For an algorithm that generates all actions at once, the space complexity is O(bm), or O(m) for an algorithm that provides actions one at a time. The time taken in real games, like chess is really high. (Russell and Norvig, n.d.)

B. Alpha-Beta Pruning

The issue with minimax search is that the number of game states it needs to look at is exponential in the depth of the tree. Unfortunately, we can't take out the exponent, however it turns out we can successfully cut it down in half. The magic is that it is possible to evaluate the correct minimax decision without checking at every node in the game tree. There is where we can an idea of pruning a large parts of tree and the technique is known as alpha-beta pruning. When applied to a standard minimax tree, it gives the same result as minimax, but alpha-beta prunes away branches that cannot possibly influence the final result.

Now considering two player game tree from figure 2, let's calculate the final optimal result. (i) Lets start the left below node with first leaf B has the value 3. 3 is assigned to min node B. (ii) Second leaf has value 12, so it would avoid and move with previous value 3. (iii) Third leaf has value 8. Even this is rejected and moves with value 3 and even as a max choice 3 at the root. (iv) The first leaf below node C has the value 2. Now, C has a value of at most 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other leafs of C. This is an example of alpha–beta pruning. (v) The first leaf below node D has the value 14, so D is worth at most 14. This is higher than MAX's value i.e. 3, so we need to keep exploring D's leaf. (vi) The second leaf of D is 5, so again we need to check. The third successor is worth 2, so now D has value exactly 2. MAX's decision at the root A is finally assigned by value of 3 as shown in figure. (Russell and Norvig, n.d.)

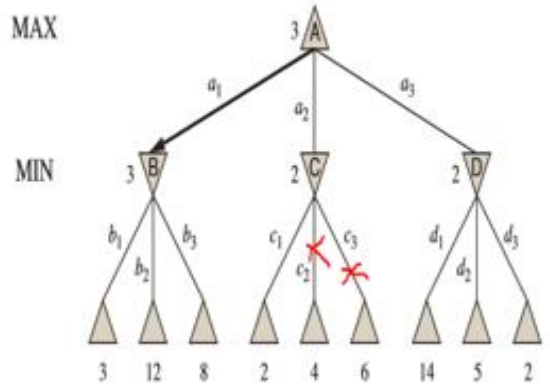


Figure. 2

In the above figure 2 MAX is alpha and MIN is beta.

This is a simple example to show how system works. Figure 3 explains how an entire branch of nodes are being cut off. Alpha beta will not just prune single node but entire branch. In exhaustive search this is very helpful.

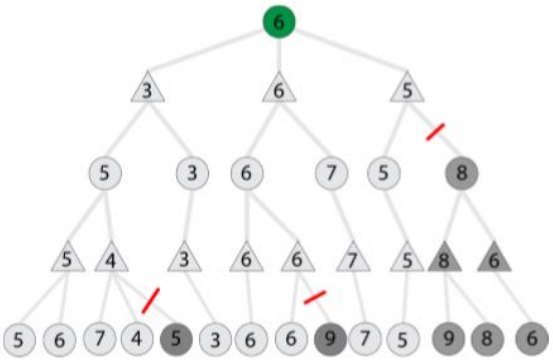


Figure 3: A large Alpha-Beta Pruned Tree

The modern chess engines can go up to a depth of 35 making trimming. For example, with 288 billion different possible positions after four moves each, we can see that being able to trim off billions of calculations is extremely advantageous. Once it reaches the depth 35 then it's impossible to evaluate entire tree directly. (P.Winston, n.d.)

The ability of pruning the branches speeds up the process by an average of 25%.

In Alpha beta pruning the worst case search time is O(bm) and best case of time is O(√bm). (D. E. Knuth, 1975)

C. Enhancements

Given a legitimate assessment work it seems that a chess engine should be able to play an ideal game. The difficulty emerges when we look at the number of moves, and by extension, decisions should have been made.

We observed how the tree is assembled and state that there are 400 different positions after every player makes one move a piece. The quicker tree traversals, with the lesser nodes visited and generated allows us to reach higher depths. The first of the enhancements is to restrain the search space. On average the first chess engine to defeat the world champion, Deep Blue, had a depth of 6-12. An engine on modern hardware will typically be limited to around a depth of 35. (P.Winston, n.d.)

The next method is iterative deepening. There is a searching technique for trees that can be utilized on any search tree data structure. It's a time management strategy which is good for depth-first searches like minimax. This is like constraining the search space as in iterative deepening the program will first assess one node, then all at level 2, trailed by level 3 and so on. This will ensure a move before the end of allotted time as, regardless of whether the full search is not completed, it can return the most recent best outcome. This is utilized in the time sensitive tournament settings with constrained calculation time per move. (greer, n.d.)

Combining iterative deepening with alpha-beta, pruning unwanted branches exploring deeply only the useful nodes. There can be issues with this, obviously, where apparently awful alternatives can result in a better choice in the long run. Only the deeper searches can fix this issue.

The last part to comprehend is that there is uneven tree development. A lot of branches will end sooner than others through the chess game completing. This makes the search time even less as it will trim a great amount of the tree so only unfinished, promising branches should be investigated even close to all of the way. (greer, n.d.)

D. New Pruning Techniques

In Tree pruning for new search methods in computer games K. Greer looks at different strategies for looking through the best move. The strategy proposed in the paper assesses and proposes different search techniques to assess a position and choose the best move.

The first method, The Chess maps Heuristic, utilizes neural networks to assess the potion and pick the move. A neural network is a collection of artificial neurons that is connected by different weighted associations. Utilizing these neurons and changing values of associations the neural network is able to keep improving through use to achieve its purpose. This networks is put to use as a positional evaluator.

This method runs numerous iterations of tests attempting each possibility and ranking the outcomes. Based on this it re-weights the associations between the neurons making it more precise each time. (Marckel, n.d.)

The system is given a position to work with and positions utilizing the following criteria: safe capture moves, safe forced moves, safe forcing moves, safe other moves, unsafe capture moves, unsafe forced moves, unsafe forcing moves, unsafe other moves. Utilizing these boundaries the potential moves are requested.

This was then joined with Alpha-Beta by utilizing the neural network to arrange the order in which nodes were arranged. This neural network is an expansion to standard Alpha-Beta and is joined with it as an upgrade. It was lightweight enough that it could be utilized in this technique but proved difficult to code such that provided any improvement over other strategies.

The new technique this paper proposes is Dynamic Move Sequences. This technique makes chains of moves from the root node instead of a branching tree like Alpha-Beta. When utilized with Heuristics to pick the most probable move it can be utilized to search deeper in several branches without expecting to take a look at different alternatives. The drawback to this is the chance of skipping over moves that Alpha-Beta would see.

This technique is proven not to return awful moves. It can regularly outperform Brute power algorithms in low depth searches, usually depths of 3-4. Because of improved innovation and therefor depth brute force algorithms, such as Alpha-Beta, are still more successful.

The test outcomes propose that the capability of the move links might be the fact that it can provide a basis for new ways to search a game tree and even under different circumstances. The coding and algorithm to implement these does not exist yet and is one of the topics that can be looked at later on. (greer, n.d.)

E. Evaluating Positions

Given Once the computer has the ability to represent the board the next stage is evaluating the position. Every node that is assessed through Alpha-Beta is ranked through this function. Recall however that every assessment the algorithm computes is not the current state but forthcoming states of the game. In a similar vein as the portrayal the technique used to store and read will enormously effect the time spent on analysis.

The first thing looked at and most significant is obviously if the game has been won, if the King has been checkmated. The following, and more significant for looking at the Alpha-Beta algorithm, is how many pieces remain and their values. Each piece has a defined worth and having more pieces, or focuses, will quite often bring about a success if two players are equivalent. Once that has been counted there are as many as different techniques as the programmer wants. Basic additions are: pawn structure, castling ability, development of pieces (their locations), and King safety. It's important to note that there are many others that are included with more or less value for each chess engine. (D. E. Knuth, 1975)

These evaluations are what Alpha-Beta uses to search with, what number is placed into the nodes. Figure 1 has only 9 nodes, with the engine seeing those as 9 values that represent the board state after 3 moves. The engine will then use those values to move according the algorithm.

The main threat in different strategies, such as, the neural networks and the dynamic move sequences referenced before, is the loss in evaluation quality because of the decrease in the search space. The potential for these strategies seems to lie in expected new alternatives for looking through a game tree. (Zhang, Liu and Wang, 2012)

F. Searching Depth

Probably the biggest reason that chess engines are improving over time is increased computing power. This permits Alpha-Beta to search more thoroughly which in turn increases its playing ability. (Ferreira, 2013)

Ferreira's research paper "The Impact of Search depth on Chess Playing Strength" is the first demo of the specific evaluations compared to depth. The reason for this paper being finding the increase in ability that each move provides. Both the software and hardware was affected when played on the Houdini Chess Engine.

This paper found that the ELO difference between each level was around 86.5 points. The Elo rating system is a strategy for figuring out the relative expertise levels of players. It is named after the creator Arpad Elo. In chess levels a rating above 2300 are generally associated with the Master title and a rating above 2500 being a Grandmaster. It should be noted

that rating alone isn't the means by which one gains these titles. Currently, FIDE has awarded 1668 players the title of "Grandmaster". Filtering out inactive players, there are 1334 GMs and 729 players with "grandmaster strength". Modern computers compete around the level of 3300. These numbers should help improving an approximation of 86.5 elo points.

As software and innovation improves there might be changes in the specific numbers discovered here. In any event, representing these progressions the essential reason that an engine improves with search depth ought to continue as before. (Ferreira, 2013)

G. Implementation

The code for this Chess game has been built using python. Python is comparatively slow with respect with other programming languages like C, C++, and Java.

In this paper, I have used minimax, alpha beta algorithms to calculate and prune the branches. When it comes to chess game I have implemented all the chess rules like (i) Castling: involves moving the king two squares towards the rook and placing the rook on the other side of king. Certain conditions should be satisfied to complete castling. (ii) En passant: When a pawn moves initially first two squares from its original place and ends the turn adjacent to a pawn of enemy's of the same rank, it might be captured by that pawn of enemy, as it had moved only one place. The move is only valid on the opponents next move immediately following the first pawn's advance. (iii) Pawn promotion: If a player reaches a pawn to opponent's eighth position then that player's pawn gets promoted to player's choice. (iv) Check: Check is said when king is under attack by at least one enemy coin. Player can move the king one square safe or obstruct other coin that makes king safe. (v) Checkmate: is a move where king is under attack by enemy coins and there is no valid move to escape.

```
if event.key == 99:  # C key
    global BOARD_COLOR
    new_colors = deepcopy(BOARD_COLORS)
    new_colors.remove(BOARD_COLOR)
    BOARD_COLOR = choice(new_colors)
    print_board(game.board, color)
```

Figure. 4

The code in this project describes the complete chess game techniques, legal moves using Alpha beta algorithm. The code explains the entire GUI and positions of each and every chess coins accurately. It has been coded to change the color of the GUI while playing or before playing. The player has to press "C" (change color) button on keyboard as shown in above code snippet in figure. 4. Also it has been taken care to evaluate the value of the coin position when the player press "E" key from keyboard as shown in figure. 5. This displays the value of the coin at that time in which position it is.

```
            if event.key == 101:  # E key
                print('eval = ' + str(Engine.evaluate_game(game) / 100))

play_as() › try › while run › for event in pygame.event.get() › if event.type == pygame.KEYDOWN

Main
"C:\Users\ashwini\PycharmProjects\Final Project Chess\venv\Scripts\python.exe" "C
pygame 1.9.6
Hello from the pygame community. https://www.pygame.org/contribute.html
eval = -0.4
```

Figure. 5

The above value 0.4 as shown in figure. 5 is evaluated for Knight Position which moved from g1 to h3. In the same way the code gives all the values for all the coins to their respective positions.

In the game, if the player plays or makes wrong move then he/she can also take it back by pressing "U" key i.e. undo, as shown below in figure. 6:

```
if event.key == 117:  # U key
    game = Engine.unmake_move(game)
    game = Engine.unmake_move(game)
    set_title(SCREEN_TITLE)
    print_board(game.board, color)
    ongoing = True
```

Figure. 6

The below code snippet figure. 7 explains only the logic of alpha beta algorithm:

```
if color == WHITE:
  for move in legal_moves(game, color):
    if verbose:
      print('\t' * depth + str(depth) + '. evaluating ' + COIN_CODES[
        get_coin(game.board, move[0])] + move2str(move))

    new_game = make_move(game, move)
    [_, score] = alpha_beta(new_game, opposing_color(color), depth - 1, alpha, beta)

    if verbose:
      print('\t' * depth + str(depth) + '. ' + str(score) + ' [{},{}]'.format(alpha, beta))

    if score == win_score(opposing_color(color)):
      return [move, score]

    if score == alpha:
      best_moves.append(move)
    if score > alpha:  # white maximizes her score
      alpha = score
      best_moves = [move]
      if alpha > beta:  # alpha-beta cutoff
        if verbose:
          print('\t' * depth + 'cutoff')
        break
  if best_moves:
    return [choice(best_moves), alpha]
  else:
    return [None, alpha]
```

Figure. 7

So, the above logic wrt white moves in alpha-beta algorithm explains: If it's a white player's turn the logic enters the 'if' loop and checks all the legal moves for a given game. If the move is valid then checks for the depth level weather its 2, 4 and so on which will be assigned initially in the code. According to the depth the algorithm moves down the tree and chooses the best value for the player in less time. Then the max value which is alpha is compared with the current score. If the alpha value is less than the score then the score value is assigned to alpha which is taken to be the best move.

The same process is continued with Black coins. But here the Black coins are compared with the beta. Here for Black

player the logic works the same way as of White. But the comparison is different way instead of alpha it is compared with beta. If the score is less than beta, the score value is assigned to beta, which is taken to be the best move for black.

H. Testing

The computers cannot simulate the human players. The real human player plays with a player on the screen (machine).

The goal of this Test case is to test whether the machine will adapt to the true human player? The initial depth ability of the machine is 2. The following screenshots demonstrate how the chess engine works.

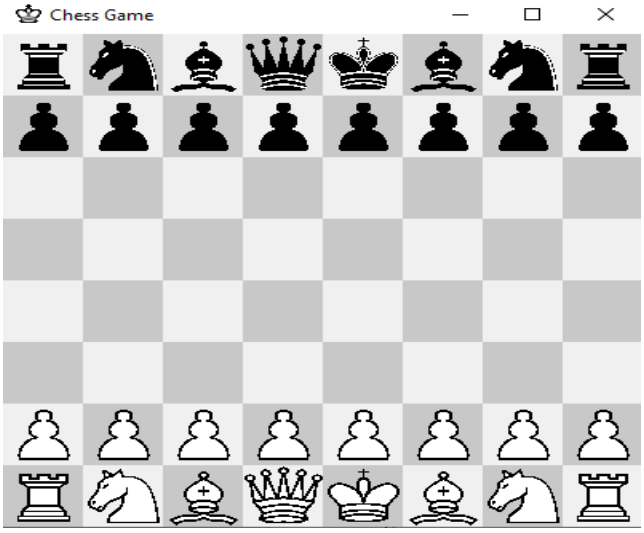


Figure. 8

Initially, both the players will arrange their coins on the board as shown in the above figure 8. Once the game starts the player (human / machine) makes the first move.

The player (human) makes the first move: d2d4. Now the opponent player (machine) also makes the move: g8f6 with the evaluation of 0.75 as shown in the below figure 9.

The player makes second move: g1f3 and the opponent makes its move d7d5 with evaluation of 0.0.

The player makes third move: c1d2 and the opponent makes its move b8c6 with evaluation of 0.4.

Figure. 9

Both the players keep on playing with the legal and appropriate moves. After a certain time and losing the coins the algorithm calculates the positions, points and provides the result who is the winner (human or machine).

The below table gives a sample of moves and evaluations from which we can assume the winner of the game.

White Player's Move	Black Player's Move	Evaluation
d2d4	g8f6	0.75
g1f3	d7d5	0.0
c1d2	b8c6	0.4
F2E1	F8A3	1.34
E1F2	G8H6	1.39
B2A3	H6F5	2.07
A3A4	F5E3	2.37
D2E3	D7D5	3.55
G2G3	C8D7	4.05
A4A5	B8C6	4.40
H2H3	C6A5	4.13
A4A5	B8C6	4.32
G3G4	O-O	5.21
G4G5	F7F6	5.66
G5F6	F8F6	4.34
H3H4	F6F1	5.32
E1F1	D7B5	5.21
C3C4	D5C4	5.27

Table. 1

IV. RESULTS

So from the above shown table 1 we can assume that White player skill level is good than the opponent Black player.

The different results mentioned below explains how the evaluations would be for different boards using different techniques.

A. Confusion Matrix

As we can observe the below given figure 10 in the confusion matrix, the Queen is usually mistaken with the bishop when they are viewed from the top. This is the only label the model has difficulty in classifying. (de Sa Delgado Neto and Mendes Campello, 2019)

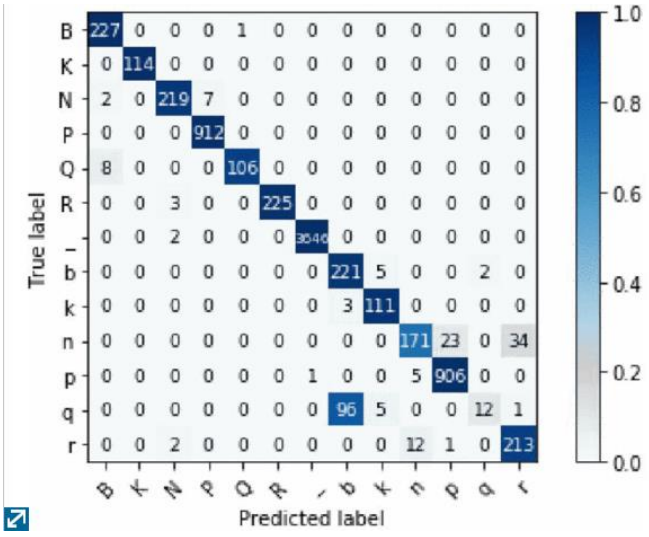


Figure. 10

B. Synthetic Board Classification

The below figure 11 shows the synthetic board classification. It results 62/64, that is roughly about 96.87% correct labels, when the empty spaces are considered, and

30/32, corresponds to 93.7% only when the valid labels are considered. In the figure red color is marked, that shows the rook and knight are mistaken (looks similar). (de Sa Delgado Neto and Mendes Campello, 2019)

Figure. 11

V. CONCLUSION

The Alpha-Beta Pruning Algorithm has been extremely successful in improving that ability of chess playing engines through its optimality. Used in the chess engine Alpha-Beta has been advancing the ability of programs for the last 60 years.

The applications of the algorithm can be expanded very easily to any zero-sum game and used to solve what otherwise would still be guessed. Until computers have the processing power to solve chess entirely, Alpha-Beta will be the top method for playing the game of chess. There can, and have been, advancements in recent years through additions and enhancements to the algorithm as well as better utilization of current and improved hardware.

VI. FUTURE WORK

The future work will investigate the potential of engaging a neural network which will assign the chess engines depending on the side of the board whether the sequence of the chess game or a position of the chess board. In addition each piece of chess could have a different value in a cluster than the one the engine itself took on. Another possibility to use the proposed system would be to allow parallelization, similar to multithread parallelization within a node, by running several instances of one engine to potential increase performance through parallelization.

The following methods gives some future improvements:

A. Coin Counter

It is practically possible for the improvement of the classification task is, the formulation of heuristic procedures in common chess games. It is unusual in the play to get a position where there are three rooks or three knights of the same color on the board as suggested by neural network in the above figure. (de Sa Delgado Neto and Mendes Campello, 2019)

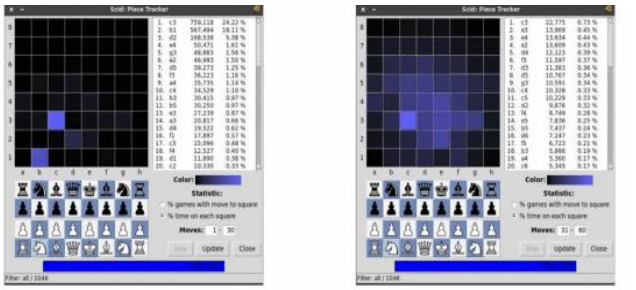
A suggestion in such situation is to presume that some knight and rook are mislabeled, in a case the probabilities which are calculated by NN should be considered by checking which knights were listed with the lowest top scores and select the right label with second highest score.

B. Using HeatMaps in Database

The progress in an algorithm that flags already identified coins as potential errors by testing whether the coin is at odd position is a good approach.

Using easily available collections of human played chess games as presented by Lichess, as of June 2019, has more than 700 million games, in which a classifier of uncommon coins positions can be handled. (lichess.org game database, 2020)

The SCID, an open source chess framework offers a coin or piece tracker tool that produces a heat map for each coin that is provided in chess database. This heatmap represented in the below given figure 12 can be used to illustrate the labelled coins which are in the unnatural positions and then, if possible using different neural networks or classification methods, perform thorough analysis to determine the correct coin mark. (Scid - Chess Database Software, 2020)



(a) Heatmap for white queen's knight in first 30 moves.

(b) Heatmap for white queen's knight in moves 31-60.

Figure. 12

C. Simulation using GAN

The below figure shows the synthetic board classification. It results 62/64, that is roughly

Additional enhancements to the synthetic data generation method will be to make the images more realistic. This could be accomplished by using a generative adversarial network popularly known as GAN, which is trained on unlabeled real chess coins data to approximate the distribution of the synthetically produced images to the distribution of real ones without much loss of details. (Shrivastava et al., 2017)

VII. REFERENCES

[1] Kang, X., Wang, Y. and Hu, Y., 2019. Research on Different Heuristics for Minimax Algorithm Insight from Connect-4 Game. Journal of Intelligent Learning Systems and Applications, 11(02), pp.15-31.

[2] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. Artificial intelligence, 6(4):293–326, 1975.

[3] P. Winston. 6. search: Games, minimax, and alpha-beta

[4] Russell, S. and Norvig, P., n.d. Artificial Intelligence.

[5] Vuckovic, V., 2012. An alternative efficient chessboard representation based on 4-bit piece coding. Yugoslav Journal of Operations Research, 22(2), pp.265-284.

[6] K. Greer. Tree pruning for new search techniques in computer games. Advances in Artificial Intelligence, 2013:2, 2013.

[7] Zhang, R., Liu, C. and Wang, C., 2012. Research on connect 6 programming based on MTD(F) and Deeper-Always Transposition Table. 2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems,.

[8] Baxter, J., n.d. A Chess Program That Learns By Combining TD(Lambda) With Game-Tree Search, In: Machine Learning.

[9] Bonet, B., n.d. “Learning Depth-First Search: A Unified Approach To Heuristic Search In Deterministic And Non-Deterministic Settings.

[10] En.wikipedia.org. 2020. Chess. [online] Available at: <https://en.wikipedia.org/wiki/Chess> [Accessed 1 September 2020]

[11] Ferreira, D., 2013. The Impact of the Search Depth on Chess Playing Strength. ICGA Journal, 36(2), pp.67-80

[12] de Sa Delgado Neto, A. and Mendes Campello, R., 2019. Chess Position Identification using Pieces Classification Based on Synthetic Images Generation and Deep Neural Network Fine-Tuning. 2019 21st Symposium on Virtual and Augmented Reality (SVR),.

[13] Database.lichess.org. 2020. Lichess.Org Game Database. [online] Available at: <https://database.lichess.org/.> [Accessed 1 September 2020].

[14] Scid.sourceforge.net. 2020. Scid - Chess Database Software. [online] Available at: <http://scid.sourceforge.net/.> [Accessed 1 September 2020].

[15] Guo, T., Qiu, H., Tong, B. and Wang, Y., 2019. Optimization and Comparison of Multiple Game Algorithms in Amazon Chess. 2019 Chinese Control And Decision Conference (CCDC),.

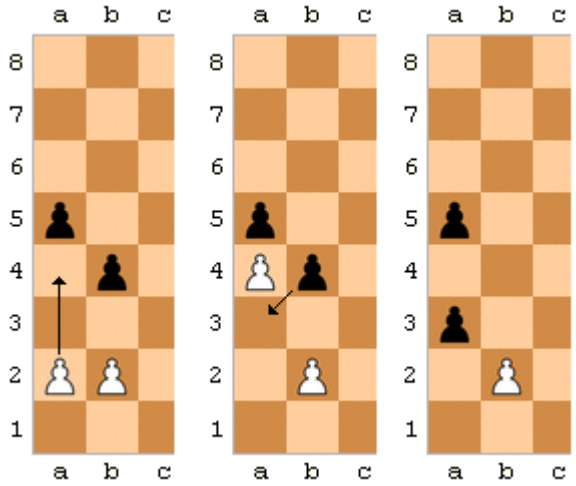
[16] Shrivastava, A., Pfister, T., Tuzel, O., Susskind, J., Wang, W. and Webb, R., 2017. Learning from Simulated and Unsupervised Images through Adversarial Training. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR),.

[17] Marckel, O., n.d. Alpha-Beta Pruning In Chess Engines.

VIII. APPENDIX

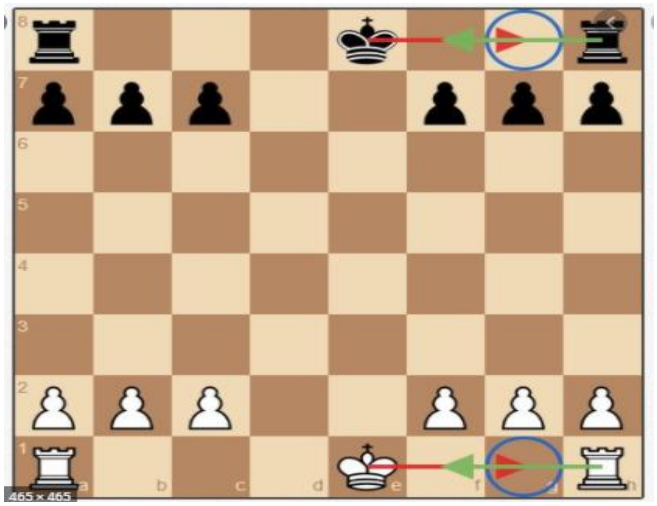
In Appendix, I will be attaching some of the screenshots of Chess game techniques which are used while playing like en passant, castling and so on.

En Passant rule: When the player’s coin faces opponent’s coin in front of it in the same path then the player can capture opponent’s coin by moving in cross path as shown in the below figure.



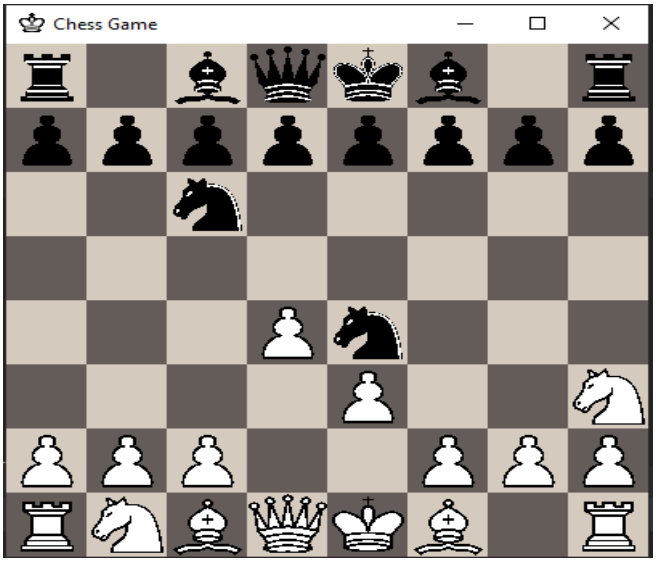
En Passant

Castling: In castling both the players will have the option to interchange the positions of King and Rook, but in between two of them there shouldn’t be any coins, below figure explains in detail



Castling

Below I am attaching the few screen shots which explains about the working of my code.



Here my coins are White and Black are machine (AI)

Above pic shows the movement of coins

With this I am also attaching my code below. Engine.py.

```
from copy import deepcopy
from random import choice
from time import sleep, time

COLOR_MASK = 1 << 3
WHITE = 0 << 3
BLACK = 1 << 3

ENDGAME_COIN_COUNT = 7

COIN_MASK = 0b111  # Binary 7 (1:2:4 =
001:010:100)
EMPTY = 0
PAWN = 1
KNIGHT = 2
BISHOP = 3
ROOK = 4
QUEEN = 5
KING = 6

COIN_TYPES = [EMPTY, PAWN, KNIGHT, BISHOP,
ROOK, QUEEN, KING]
COIN_VALUES = {EMPTY: 0, PAWN: 100, KNIGHT:
300, BISHOP: 300, ROOK: 500, QUEEN: 900,
KING: 42000}

FILES = ['a', 'b', 'c', 'd', 'e', 'f', 'g',
'h']
RANKS = ['1', '2', '3', '4', '5', '6', '7',
'8']

CASTLE_KINGSIDE_WHITE = 0b1 << 0
CASTLE_QUEENSIDE_WHITE = 0b1 << 1
CASTLE_KINGSIDE_BLACK = 0b1 << 2
CASTLE_QUEENSIDE_BLACK = 0b1 << 3

FULL_CASTLING_RIGHTS = CASTLE_KINGSIDE_WHITE
| CASTLE_QUEENSIDE_WHITE |
CASTLE_KINGSIDE_BLACK |
CASTLE_QUEENSIDE_BLACK

# COIN AND COLOR BITMASK = 0bCPPP. Cap C for
color and small c for coins. Here the pgm
uses little endian rank file

ALL_SQUARES = 0xFFFFFFFFFFFFFFFF  # Access
violation reading location in hexadecimal,
18446744073709551615 decimal
FILE_A = 0x0101010101010101  #
72340172838076673 decimal,
1000000010000000100000001000000010000000100
0000100000001 bin
FILE_B = 0x0202020202020202  #
144680345676153346
FILE_C = 0x0404040404040404  #
289360691352306692
FILE_D = 0x0808080808080808
FILE_E = 0x1010101010101010
FILE_F = 0x2020202020202020
FILE_G = 0x4040404040404040
FILE_H = 0x8080808080808080
RANK_1 = 0x00000000000000FF  # 255, 11111111
binary
RANK_2 = 0x000000000000FF00  # 65280
RANK_3 = 0x0000000000FF0000
RANK_4 = 0x00000000FF000000
RANK_5 = 0x000000FF00000000
RANK_6 = 0x0000FF0000000000
RANK_7 = 0x00FF000000000000
RANK_8 = 0xFF00000000000000
DIAG_A1H8 = 0x8040201008040201
ANTI_DIAG_H1A8 = 0x0102040810204080
LIGHT_SQUARES = 0x55AA55AA55AA55AA
DARK_SQUARES = 0xAA55AA55AA55AA55

FILE_MASKS = [FILE_A, FILE_B, FILE_C,
FILE_D, FILE_E, FILE_F, FILE_G, FILE_H]
RANK_MASKS = [RANK_1, RANK_2, RANK_3,
RANK_4, RANK_5, RANK_6, RANK_7, RANK_8]

# bit_boards = 0b(h8)(g8)...(b1)(a1)
# board = [ a1, b1, ..., g8, h8 ]

INITIAL_BOARD = [ROOK | WHITE, KNIGHT |
WHITE, BISHOP | WHITE, QUEEN | WHITE, KING |
WHITE, BISHOP | WHITE,
                 KNIGHT | WHITE, ROOK |
WHITE,
                 PAWN | WHITE, PAWN | WHITE,
PAWN | WHITE, PAWN | WHITE, PAWN | WHITE,
PAWN | WHITE,
                 PAWN | WHITE, PAWN | WHITE,
                 EMPTY, EMPTY, EMPTY, EMPTY,
EMPTY, EMPTY, EMPTY, EMPTY,
                 EMPTY, EMPTY, EMPTY, EMPTY,
EMPTY, EMPTY, EMPTY, EMPTY,
                 EMPTY, EMPTY, EMPTY, EMPTY,
EMPTY, EMPTY, EMPTY, EMPTY,
                 EMPTY, EMPTY, EMPTY, EMPTY,
EMPTY, EMPTY, EMPTY, EMPTY,
                 PAWN | BLACK, PAWN | BLACK,
PAWN | BLACK, PAWN | BLACK, PAWN | BLACK,
PAWN | BLACK,
                 PAWN | BLACK, PAWN | BLACK,
                 ROOK | BLACK, KNIGHT |
BLACK, BISHOP | BLACK, QUEEN | BLACK, KING |
BLACK, BISHOP | BLACK,
                 KNIGHT | BLACK, ROOK |
BLACK]

EMPTY_BOARD = [EMPTY for _ in range(64)]

INITIAL_FEN =
'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR
w KQkq - 0 1'
STROKES_YOLO =
'1k6/2b1p3/Qp4N1/4r2P/2B2q2/1R6/2Pn2K1/8 w -
- 0 1'

COIN_CODES = {KING | WHITE: 'K',
```

```
                QUEEN | WHITE: 'Q',
                ROOK | WHITE: 'R',
                BISHOP | WHITE: 'B',
                KNIGHT | WHITE: 'N',
                PAWN | WHITE: 'P',
                KING | BLACK: 'k',
                QUEEN | BLACK: 'q',
                ROOK | BLACK: 'r',
                BISHOP | BLACK: 'b',
                KNIGHT | BLACK: 'n',
                PAWN | BLACK: 'p',
                EMPTY: '.'}
COIN_CODES.update({v: k for k, v in
COIN_CODES.items()})

DOUBLED_PAWN_PENALTY = 5
ISOLATED_PAWN_PENALTY = 10
BACKWARDS_PAWN_PENALTY = 4
PASSED_PAWN_TABLE = 10
ROOK_SEMI_OPEN_FILE_TABLE = 5
ROOK_OPEN_FILE_TABLE = 8
ROOK_ON_SEVENTH_TABLE = 10

PAWN_TABLE = [0, 0, 0, 0, 0, 0, 0, 0,
              50, 50, 50, 50, 50, 50, 50,
50,
              10, 10, 20, 30, 30, 20, 10,
10,
              5, 5, 10, 25, 25, 10, 5, 5,
              0, 0, 0, 20, 20, 0, 0, 0,
              5, -5, -10, 0, 0, -10, -5, 5,
              5, 10, 10, -20, -20, 10, 10,
5,
              0, 0, 0, 0, 0, 0, 0, 0]

KNIGHT_TABLE = [-50, -40, -30, -30, -30, -
30, -40, -50,
                -40, -20, 0, 0, 0, 0, -20, -
40,
                -30, 0, 10, 15, 15, 10, 0, -
30,
                -30, 5, 15, 20, 20, 15, 5, -
30,
                -30, 0, 15, 20, 20, 15, 0, -
30,
                -30, 5, 10, 15, 15, 10, 5, -
30,
                -40, -20, 0, 5, 5, 0, -20, -
40,
                -50, -90, -30, -30, -30, -
30, -90, -50]

BISHOP_TABLE = [-20, -10, -10, -10, -10, -
10, -10, -20,
                -10, 0, 0, 0, 0, 0, 0, -10,
                -10, 0, 5, 10, 10, 5, 0, -
10,
                -10, 5, 5, 10, 10, 5, 5, -
10,
                -10, 0, 10, 10, 10, 10, 0, -
10,
                -10, 10, 10, 10, 10, 10, 10,
-10,
                -10, 5, 0, 0, 0, 0, 5, -10,
                -20, -10, -90, -10, -10, -
90, -10, -20]

KING_TABLE = [-30, -40, -40, -50, -50, -40,
-40, -30,
              -30, -40, -40, -50, -50, -40,
-40, -30,
              -30, -40, -40, -50, -50, -40,
-40, -30,
              -30, -40, -40, -50, -50, -40,
-40, -30,
              -20, -30, -30, -40, -40, -30,
-30, -20,
              -10, -20, -20, -20, -20, -20,
-20, -10,
              20, 20, 0, 0, 0, 0, 20, 20,
              20, 30, 10, 0, 0, 10, 30, 20]

KING_ENDGAME_TABLE = [-50, -40, -30, -20, -
20, -30, -40, -50,
                      -30, -20, -10, 0, 0, -
10, -20, -30,
                      -30, -10, 20, 30, 30,
20, -10, -30,
                      -30, -10, 30, 40, 40,
30, -10, -30,
                      -30, -10, 30, 40, 40,
30, -10, -30,
                      -30, -10, 20, 30, 30,
20, -10, -30,
                      -30, -30, 0, 0, 0, 0,
-30, -30,
                      -50, -30, -30, -30, -
30, -30, -30, -50]

verbose = False


# ========== CHESS GAME ==========

class Game:
    def __init__(self, FEN=''):
        self.board = INITIAL_BOARD
        self.to_move = WHITE
        self.ep_square = 0
        self.castling_rights =
FULL_CASTLING_RIGHTS  # CASTLING BITMASK =
0bqkQK
        self.halfmove_clock = 0
        self.fullmove_number = 1

        self.position_history = []
        if FEN != '':
            self.load_FEN(FEN)

self.position_history.append(FEN)
        else:
```

```
self.position_history.append(INITIAL_FEN)

        self.move_history = []

    def get_move_list(self):
        return ' '.join(self.move_history)

    def to_FEN(self):
        FEN_str = ''

        for i in range(len(RANKS)):
            first = len(self.board) - 8 * (i
+ 1)
            empty_sqrs = 0
            for file in range(len(FILES)):
                coin = self.board[first +
file]
                if coin & COIN_MASK ==
EMPTY:
                    empty_sqrs += 1
                else:
                    if empty_sqrs > 0:
                        FEN_str +=
'{}'.format(empty_sqrs)
                    FEN_str +=
'{}'.format(coin2str(coin))
                    empty_sqrs = 0
            if empty_sqrs > 0:
                FEN_str +=
'{}'.format(empty_sqrs)
            FEN_str += '/'
        FEN_str = FEN_str[:-1] + ' '

        if self.to_move == WHITE:
            FEN_str += 'w '
        if self.to_move == BLACK:
            FEN_str += 'b '

        if self.castling_rights &
CASTLE_KINGSIDE_WHITE: # castling code K
            FEN_str += 'K'
        if self.castling_rights &
CASTLE_QUEENSIDE_WHITE: # castling code Q
            FEN_str += 'Q'
        if self.castling_rights &
CASTLE_KINGSIDE_BLACK: # castling code k
            FEN_str += 'k'
        if self.castling_rights &
CASTLE_QUEENSIDE_BLACK: # castling code q
            FEN_str += 'q'
        if self.castling_rights == 0:
            FEN_str += '-'
        FEN_str += ' '

        if self.ep_square == 0:
            FEN_str += '-'
        else:
            FEN_str +=
bb2str(self.ep_square)

        FEN_str += '
{}'.format(self.halfmove_clock)
        FEN_str += '
{}'.format(self.fullmove_number)
        return FEN_str

    def load_FEN(self, FEN_str):
        FEN_list = FEN_str.split(' ')

        board_str = FEN_list[0]
        rank_list = board_str.split('/')
        rank_list.reverse()
        self.board = []

        for rank in rank_list:
            rank_coins = []
            for p in rank:
                if p.isdigit():
                    for _ in range(int(p)):

rank_coins.append(EMPTY)
                else:

rank_coins.append(str2coin(p))
            self.board.extend(rank_coins)

        to_move_str = FEN_list[1].lower()
        if to_move_str == 'w':
            self.to_move = WHITE
        if to_move_str == 'b':
            self.to_move = BLACK

        castling_rights_str = FEN_list[2]
        self.castling_rights = 0
        if castling_rights_str.find('K') >=
0:
            self.castling_rights |=
CASTLE_KINGSIDE_WHITE
        if castling_rights_str.find('Q') >=
0:
            self.castling_rights |=
CASTLE_QUEENSIDE_WHITE
        if castling_rights_str.find('k') >=
0:
            self.castling_rights |=
CASTLE_KINGSIDE_BLACK
        if castling_rights_str.find('q') >=
0:
            self.castling_rights |=
CASTLE_QUEENSIDE_BLACK

        ep_str = FEN_list[3]
        if ep_str == '-':
            self.ep_square = 0
        else:
            self.ep_square = str2bb(ep_str)

        self.halfmove_clock =
int(FEN_list[4])
        self.fullmove_number =
int(FEN_list[5])
```

```
# ================================


def get_coin(board, bitboard):
    return board[bb2index(bitboard)]


def bb2index(bitboard):
    for i in range(64):
        if bitboard & (0b1 << i):
            return i


def str2index(position_str):
    file =
FILES.index(position_str[0].lower())
    rank = RANKS.index(position_str[1])
    return 8 * rank + file


def bb2str(bitboard):
    for i in range(64):
        if bitboard & (0b1 << i):
            file = i % 8
            rank = int(i / 8)
            return
'{}{}'.format(FILES[file], RANKS[rank])


def str2bb(position_str):
    return 0b1 << str2index(position_str)


def move2str(move):
    return bb2str(move[0]) + bb2str(move[1])


def single_gen(bitboard):
    for i in range(64):
        bit = 0b1 << i
        if bitboard & bit:
            yield bit


def coin_gen(board, coin_code):
    for i in range(64):
        if board[i] & COIN_MASK ==
coin_code:
            yield 0b1 << i


def colored_coin_gen(board, coin_code,
color):
    for i in range(64):
        if board[i] == coin_code | color:
            yield 0b1 << i


def opposing_color(color):
    if color == WHITE:
        return BLACK
    if color == BLACK:
        return WHITE


def coin2str(coin):
    return COIN_CODES[coin]


def str2coin(string):
    return COIN_CODES[string]


def print_board(board):
    print('')
    for i in range(len(RANKS)):
        rank_str = str(8 - i) + ' '
        first = len(board) - 8 * (i + 1)
        for file in range(len(FILES)):
            rank_str += '{}
'.format(coin2str(board[first + file]))
        print(rank_str)
    print('  a b c d e f g h')


def print_rotated_board(board):
    r_board = rotate_board(board)
    print('')
    for i in range(len(RANKS)):
        rank_str = str(i + 1) + ' '
        first = len(r_board) - 8 * (i + 1)
        for file in range(len(FILES)):
            rank_str += '{}
'.format(coin2str(r_board[first + file]))
        print(rank_str)
    print('  h g f e d c b a')


def print_bitboard(bitboard):
    print('')
    for rank in range(len(RANKS)):
        rank_str = str(8 - rank) + ' '
        for file in range(len(FILES)):
            if (bitboard >> (file + (7 -
rank) * 8)) & 0b1:
                rank_str += '# '
            else:
                rank_str += '. '
        print(rank_str)
    print('  a b c d e f g h')


def lsb(bitboard):          # least
significant bit
    for i in range(64):
        bit = (0b1 << i)
        if bit & bitboard:
            return bit


def msb(bitboard):          # most
```

```
significant bit
    for i in range(64):
        bit = (0b1 << (63 - i))
        if bit & bitboard:
            return bit


def get_colored_coins(board, color):
    return list2int([(i != EMPTY and i &
COLOR_MASK == color) for i in board])


def empty_squares(board):
    return list2int([i == EMPTY for i in
board])


def occupied_squares(board):
    return nnot(empty_squares(board))


def list2int(lst):
    rev_list = lst[:]
    rev_list.reverse()
    return int('0b' + ''.join(['1' if i else
'0' for i in rev_list]), 2)


def nnot(bitboard):
    return ~bitboard & ALL_SQUARES


def rotate_board(board):
    rotated_board = deepcopy(board)
    rotated_board.reverse()
    return rotated_board


def flip_board_v(board):
    flip = [56, 57, 58, 59, 60, 61, 62, 63,
            48, 49, 50, 51, 52, 53, 54, 55,
            40, 41, 42, 43, 44, 45, 46, 47,
            32, 33, 34, 35, 36, 37, 38, 39,
            24, 25, 26, 27, 28, 29, 30, 31,
            16, 17, 18, 19, 20, 21, 22, 23,
            8, 9, 10, 11, 12, 13, 14, 15,
            0, 1, 2, 3, 4, 5, 6, 7]

    return deepcopy([board[flip[i]] for i in
range(64)])


def east_one(bitboard):
    return (bitboard << 1) & nnot(FILE_A)


def west_one(bitboard):
    return (bitboard >> 1) & nnot(FILE_H)


def north_one(bitboard):
    return (bitboard << 8) & nnot(RANK_1)


def south_one(bitboard):
    return (bitboard >> 8) & nnot(RANK_8)


def NE_one(bitboard):
    return north_one(east_one(bitboard))


def NW_one(bitboard):
    return north_one(west_one(bitboard))


def SE_one(bitboard):
    return south_one(east_one(bitboard))


def SW_one(bitboard):
    return south_one(west_one(bitboard))


def move_coin(board, move):
    new_board = deepcopy(board)
    new_board[bb2index(move[1])] =
new_board[bb2index(move[0])]
    new_board[bb2index(move[0])] = EMPTY
    return new_board


def make_move(game, move):
    new_game = deepcopy(game)
    leaving_position = move[0]
    arriving_position = move[1]

    # update_clocks
    new_game.halfmove_clock += 1
    if new_game.to_move == BLACK:
        new_game.fullmove_number += 1

    # reset clock if capture
    if get_coin(new_game.board,
arriving_position) != EMPTY:
        new_game.halfmove_clock = 0

    # for pawns: reset clock, removed
captured ep, set new ep, promote
    if get_coin(new_game.board,
leaving_position) & COIN_MASK == PAWN:
        new_game.halfmove_clock = 0

        if arriving_position ==
game.ep_square:
            new_game.board =
remove_captured_ep(new_game)

        if is_double_push(leaving_position,
arriving_position):
            new_game.ep_square =
new_ep_square(leaving_position)
```

```
        if arriving_position & (RANK_1 |
RANK_8):

new_game.board[bb2index(leaving_position)] =
new_game.to_move | QUEEN

    # reset ep square if not updated
    if new_game.ep_square == game.ep_square:
        new_game.ep_square = 0

    # update castling rights for rook moves
    if leaving_position == str2bb('a1'):
        new_game.castling_rights =
remove_castling_rights(new_game,
CASTLE_QUEENSIDE_WHITE)
    if leaving_position == str2bb('h1'):
        new_game.castling_rights =
remove_castling_rights(new_game,
CASTLE_KINGSIDE_WHITE)
    if leaving_position == str2bb('a8'):
        new_game.castling_rights =
remove_castling_rights(new_game,
CASTLE_QUEENSIDE_BLACK)
    if leaving_position == str2bb('h8'):
        new_game.castling_rights =
remove_castling_rights(new_game,
CASTLE_KINGSIDE_BLACK)

    # castling
    if get_coin(new_game.board,
leaving_position) == WHITE | KING:
        new_game.castling_rights =
remove_castling_rights(new_game,
CASTLE_KINGSIDE_WHITE |
CASTLE_QUEENSIDE_WHITE)
        if leaving_position == str2bb('e1'):
            if arriving_position ==
str2bb('g1'):
                new_game.board =
move_coin(new_game.board, [str2bb('h1'),
str2bb('f1')])
            if arriving_position ==
str2bb('c1'):
                new_game.board =
move_coin(new_game.board, [str2bb('a1'),
str2bb('d1')])

    if get_coin(new_game.board,
leaving_position) == BLACK | KING:
        new_game.castling_rights =
remove_castling_rights(new_game,
CASTLE_KINGSIDE_BLACK |
CASTLE_QUEENSIDE_BLACK)
        if leaving_position == str2bb('e8'):
            if arriving_position ==
str2bb('g8'):
                new_game.board =
move_coin(new_game.board, [str2bb('h8'),
str2bb('f8')])
            if arriving_position ==
str2bb('c8'):
                new_game.board =
move_coin(new_game.board, [str2bb('a8'),
str2bb('d8')])

    # update positions and next to move
    new_game.board =
move_coin(new_game.board, (leaving_position,
arriving_position))
    new_game.to_move =
opposing_color(new_game.to_move)

    # update history

new_game.move_history.append(move2str(move))

new_game.position_history.append(new_game.to
_FEN())
    return new_game


def unmake_move(game):
    if len(game.position_history) < 2:
        return deepcopy(game)

    new_game = Game(game.position_history[-
2])
    new_game.move_history =
deepcopy(game.move_history)[:-1]
    new_game.position_history =
deepcopy(game.position_history)[:-1]
    return new_game


def get_rank(rank_num):
    rank_num = int(rank_num)
    return RANK_MASKS[rank_num]


def get_file(file_str):
    file_str = file_str.lower()
    file_num = FILES.index(file_str)
    return FILE_MASKS[file_num]


def get_filter(filter_str):
    if filter_str in FILES:
        return get_file(filter_str)
    if filter_str in RANKS:
        return get_rank(filter_str)


# ========== PAWN ==========

def get_all_pawns(board):
    return list2int([i & COIN_MASK == PAWN
for i in board])


def get_pawns(board, color):
    return list2int([i == color | PAWN for i
```

```
in board])


def pawn_moves(moving_coin, game, color):
    return pawn_pushes(moving_coin,
game.board, color) |
pawn_captures(moving_coin, game, color)


def pawn_captures(moving_coin, game, color):
    return pawn_simple_captures(moving_coin,
game, color) | pawn_ep_captures(moving_coin,
game, color)


def pawn_pushes(moving_coin, board, color):
    return pawn_simple_pushes(moving_coin,
board, color) |
pawn_double_pushes(moving_coin, board,
color)


def pawn_simple_captures(attacking_coin,
game, color):
    return pawn_attacks(attacking_coin,
game.board, color) &
get_colored_coins(game.board,
opposing_color(color))


def pawn_ep_captures(attacking_coin, game,
color):
    if color == WHITE:
        ep_squares = game.ep_square & RANK_6
    if color == BLACK:
        ep_squares = game.ep_square & RANK_3
    return pawn_attacks(attacking_coin,
game.board, color) & ep_squares


def pawn_attacks(attacking_coin, board,
color):
    return pawn_east_attacks(attacking_coin,
board, color) |
pawn_west_attacks(attacking_coin, board,
color)


def pawn_simple_pushes(moving_coin, board,
color):
    if color == WHITE:
        return north_one(moving_coin) &
empty_squares(board)
    if color == BLACK:
        return south_one(moving_coin) &
empty_squares(board)


def pawn_double_pushes(moving_coin, board,
color):
    if color == WHITE:
        return
north_one(pawn_simple_pushes(moving_coin,
board, color)) & (empty_squares(board) &
RANK_4)
    if color == BLACK:
        return
south_one(pawn_simple_pushes(moving_coin,
board, color)) & (empty_squares(board) &
RANK_5)


def pawn_east_attacks(attacking_coin, board,
color):
    if color == WHITE:
        return NE_one(attacking_coin &
get_colored_coins(board, color))
    if color == BLACK:
        return SE_one(attacking_coin &
get_colored_coins(board, color))


def pawn_west_attacks(attacking_coin, board,
color):
    if color == WHITE:
        return NW_one(attacking_coin &
get_colored_coins(board, color))
    if color == BLACK:
        return SW_one(attacking_coin &
get_colored_coins(board, color))


def pawn_double_attacks(attacking_coin,
board, color):
    return pawn_east_attacks(attacking_coin,
board, color) &
pawn_west_attacks(attacking_coin, board,
color)


def is_double_push(leaving_square,
target_square):
    return (leaving_square & RANK_2 and
target_square & RANK_4) or \
           (leaving_square & RANK_7 and
target_square & RANK_5)


def new_ep_square(leaving_square):
    if leaving_square & RANK_2:
        return north_one(leaving_square)
    if leaving_square & RANK_7:
        return south_one(leaving_square)


def remove_captured_ep(game):
    new_board = deepcopy(game.board)
    if game.ep_square & RANK_3:

new_board[bb2index(north_one(game.ep_square)
)] = EMPTY
    if game.ep_square & RANK_6:
```

```
new_board[bb2index(south_one(game.ep_square)
)] = EMPTY
    return new_board


# ========== KNIGHT ==========

def get_knights(board, color):
    return list2int([i == color | KNIGHT for
i in board])


def knight_moves(moving_coin, board, color):
    return knight_attacks(moving_coin) &
nnot(get_colored_coins(board, color))


def knight_attacks(moving_coin):
    return knight_NNE(moving_coin) | \
           knight_ENE(moving_coin) | \
           knight_NNW(moving_coin) | \
           knight_WNW(moving_coin) | \
           knight_SSE(moving_coin) | \
           knight_ESE(moving_coin) | \
           knight_SSW(moving_coin) | \
           knight_WSW(moving_coin)


def knight_WNW(moving_coin):
    return moving_coin << 6 & nnot(FILE_G |
FILE_H)


def knight_ENE(moving_coin):
    return moving_coin << 10 & nnot(FILE_A |
FILE_B)


def knight_NNW(moving_coin):
    return moving_coin << 15 & nnot(FILE_H)


def knight_NNE(moving_coin):
    return moving_coin << 17 & nnot(FILE_A)


def knight_ESE(moving_coin):
    return moving_coin >> 6 & nnot(FILE_A |
FILE_B)


def knight_WSW(moving_coin):
    return moving_coin >> 10 & nnot(FILE_G |
FILE_H)


def knight_SSE(moving_coin):
    return moving_coin >> 15 & nnot(FILE_A)


def knight_SSW(moving_coin):
    return moving_coin >> 17 & nnot(FILE_H)


def knight_fill(moving_coin, n):
    fill = moving_coin
    for _ in range(n):
        fill |= knight_attacks(fill)
    return fill


def knight_distance(pos1, pos2):
    init_bitboard = str2bb(pos1)
    end_bitboard = str2bb(pos2)
    fill = init_bitboard
    dist = 0
    while fill & end_bitboard == 0:
        dist += 1
        fill = knight_fill(init_bitboard,
dist)
    return dist


# ========== KING ==========

def get_king(board, color):
    return list2int([i == color | KING for i
in board])


def king_moves(moving_coin, board, color):
    return king_attacks(moving_coin) &
nnot(get_colored_coins(board, color))


def king_attacks(moving_coin):
    king_atks = moving_coin |
east_one(moving_coin) |
west_one(moving_coin)
    king_atks |= north_one(king_atks) |
south_one(king_atks)
    return king_atks & nnot(moving_coin)


def can_castle_kingside(game, color):
    if color == WHITE:
        return (game.castling_rights &
CASTLE_KINGSIDE_WHITE) and \
               game.board[str2index('f1')]
== EMPTY and \
               game.board[str2index('g1')]
== EMPTY and \
               (not
is_attacked(str2bb('e1'), game.board,
opposing_color(color))) and \
               (not
is_attacked(str2bb('f1'), game.board,
opposing_color(color))) and \
               (not
is_attacked(str2bb('g1'), game.board,
opposing_color(color)))
```

```
    if color == BLACK:
        return (game.castling_rights &
CASTLE_KINGSIDE_BLACK) and \
               game.board[str2index('f8')]
== EMPTY and \
               game.board[str2index('g8')]
== EMPTY and \
               (not
is_attacked(str2bb('e8'), game.board,
opposing_color(color))) and \
               (not
is_attacked(str2bb('f8'), game.board,
opposing_color(color))) and \
               (not
is_attacked(str2bb('g8'), game.board,
opposing_color(color)))


def can_castle_queenside(game, color):
    if color == WHITE:
        return (game.castling_rights &
CASTLE_QUEENSIDE_WHITE) and \
               game.board[str2index('b1')]
== EMPTY and \
               game.board[str2index('c1')]
== EMPTY and \
               game.board[str2index('d1')]
== EMPTY and \
               (not
is_attacked(str2bb('c1'), game.board,
opposing_color(color))) and \
               (not
is_attacked(str2bb('d1'), game.board,
opposing_color(color))) and \
               (not
is_attacked(str2bb('e1'), game.board,
opposing_color(color)))
    if color == BLACK:
        return (game.castling_rights &
CASTLE_QUEENSIDE_BLACK) and \
               game.board[str2index('b8')]
== EMPTY and \
               game.board[str2index('c8')]
== EMPTY and \
               game.board[str2index('d8')]
== EMPTY and \
               (not
is_attacked(str2bb('c8'), game.board,
opposing_color(color))) and \
               (not
is_attacked(str2bb('d8'), game.board,
opposing_color(color))) and \
               (not
is_attacked(str2bb('e8'), game.board,
opposing_color(color)))


def castle_kingside_move(game):
    if game.to_move == WHITE:
        return str2bb('e1'), str2bb('g1')
    if game.to_move == BLACK:
        return str2bb('e8'), str2bb('g8')


def castle_queenside_move(game):
    if game.to_move == WHITE:
        return str2bb('e1'), str2bb('c1')
    if game.to_move == BLACK:
        return str2bb('e8'), str2bb('c8')


def remove_castling_rights(game,
removed_rights):
    return game.castling_rights &
~removed_rights


# ========== BISHOP ==========

def get_bishops(board, color):
    return list2int([i == color | BISHOP for
i in board])


def bishop_rays(moving_coin):
    return diagonal_rays(moving_coin) |
anti_diagonal_rays(moving_coin)


def diagonal_rays(moving_coin):
    return NE_ray(moving_coin) |
SW_ray(moving_coin)


def anti_diagonal_rays(moving_coin):
    return NW_ray(moving_coin) |
SE_ray(moving_coin)


def NE_ray(moving_coin):
    ray_atks = NE_one(moving_coin)
    for _ in range(6):
        ray_atks |= NE_one(ray_atks)
    return ray_atks & ALL_SQUARES


def SE_ray(moving_coin):
    ray_atks = SE_one(moving_coin)
    for _ in range(6):
        ray_atks |= SE_one(ray_atks)
    return ray_atks & ALL_SQUARES


def NW_ray(moving_coin):
    ray_atks = NW_one(moving_coin)
    for _ in range(6):
        ray_atks |= NW_one(ray_atks)
    return ray_atks & ALL_SQUARES


def SW_ray(moving_coin):
    ray_atks = SW_one(moving_coin)
```

```
    for _ in range(6):
        ray_atks |= SW_one(ray_atks)
    return ray_atks & ALL_SQUARES


def NE_attacks(single_coin, board, color):
    blocker = lsb(NE_ray(single_coin) &
occupied_squares(board))
    if blocker:
        return NE_ray(single_coin) ^
NE_ray(blocker)
    else:
        return NE_ray(single_coin)


def NW_attacks(single_coin, board, color):
    blocker = lsb(NW_ray(single_coin) &
occupied_squares(board))
    if blocker:
        return NW_ray(single_coin) ^
NW_ray(blocker)
    else:
        return NW_ray(single_coin)


def SE_attacks(single_coin, board, color):
    blocker = msb(SE_ray(single_coin) &
occupied_squares(board))
    if blocker:
        return SE_ray(single_coin) ^
SE_ray(blocker)
    else:
        return SE_ray(single_coin)


def SW_attacks(single_coin, board, color):
    blocker = msb(SW_ray(single_coin) &
occupied_squares(board))
    if blocker:
        return SW_ray(single_coin) ^
SW_ray(blocker)
    else:
        return SW_ray(single_coin)


def diagonal_attacks(single_coin, board,
color):
    return NE_attacks(single_coin, board,
color) | SW_attacks(single_coin, board,
color)


def anti_diagonal_attacks(single_coin,
board, color):
    return NW_attacks(single_coin, board,
color) | SE_attacks(single_coin, board,
color)


def bishop_attacks(moving_coin, board,
color):
    atks = 0
    for coin in single_gen(moving_coin):
        atks |= diagonal_attacks(coin,
board, color) | anti_diagonal_attacks(coin,
board, color)
    return atks


def bishop_moves(moving_coin, board, color):
    return bishop_attacks(moving_coin,
board, color) &
nnot(get_colored_coins(board, color))


# ========== ROOK ==========

def get_rooks(board, color):
    return list2int([i == color | ROOK for i
in board])


def rook_rays(moving_coin):
    return rank_rays(moving_coin) |
file_rays(moving_coin)


def rank_rays(moving_coin):
    return east_ray(moving_coin) |
west_ray(moving_coin)


def file_rays(moving_coin):
    return north_ray(moving_coin) |
south_ray(moving_coin)


def east_ray(moving_coin):
    ray_atks = east_one(moving_coin)
    for _ in range(6):
        ray_atks |= east_one(ray_atks)
    return ray_atks & ALL_SQUARES


def west_ray(moving_coin):
    ray_atks = west_one(moving_coin)
    for _ in range(6):
        ray_atks |= west_one(ray_atks)
    return ray_atks & ALL_SQUARES


def north_ray(moving_coin):
    ray_atks = north_one(moving_coin)
    for _ in range(6):
        ray_atks |= north_one(ray_atks)
    return ray_atks & ALL_SQUARES


def south_ray(moving_coin):
    ray_atks = south_one(moving_coin)
    for _ in range(6):
        ray_atks |= south_one(ray_atks)
```

```
    return ray_atks & ALL_SQUARES


def east_attacks(single_coin, board, color):
    blocker = lsb(east_ray(single_coin) &
occupied_squares(board))
    if blocker:
        return east_ray(single_coin) ^
east_ray(blocker)
    else:
        return east_ray(single_coin)


def west_attacks(single_coin, board, color):
    blocker = msb(west_ray(single_coin) &
occupied_squares(board))
    if blocker:
        return west_ray(single_coin) ^
west_ray(blocker)
    else:
        return west_ray(single_coin)


def rank_attacks(single_coin, board, color):
    return east_attacks(single_coin, board,
color) | west_attacks(single_coin, board,
color)


def north_attacks(single_coin, board,
color):
    blocker = lsb(north_ray(single_coin) &
occupied_squares(board))
    if blocker:
        return north_ray(single_coin) ^
north_ray(blocker)
    else:
        return north_ray(single_coin)


def south_attacks(single_coin, board,
color):
    blocker = msb(south_ray(single_coin) &
occupied_squares(board))
    if blocker:
        return south_ray(single_coin) ^
south_ray(blocker)
    else:
        return south_ray(single_coin)


def file_attacks(single_coin, board, color):
    return north_attacks(single_coin, board,
color) | south_attacks(single_coin, board,
color)


def rook_attacks(moving_coin, board, color):
    atks = 0
    for single_coin in
single_gen(moving_coin):
        atks |= rank_attacks(single_coin,
board, color) | file_attacks(single_coin,
board, color)
    return atks


def rook_moves(moving_coin, board, color):
    return rook_attacks(moving_coin, board,
color) & nnot(get_colored_coins(board,
color))


# ========== QUEEN ==========

def get_queen(board, color):
    return list2int([i == color | QUEEN for
i in board])


def queen_rays(moving_coin):
    return rook_rays(moving_coin) |
bishop_rays(moving_coin)


def queen_attacks(moving_coin, board,
color):
    return bishop_attacks(moving_coin,
board, color) | rook_attacks(moving_coin,
board, color)


def queen_moves(moving_coin, board, color):
    return bishop_moves(moving_coin, board,
color) | rook_moves(moving_coin, board,
color)


def is_attacked(target, board,
attacking_color):
    return count_attacks(target, board,
attacking_color) > 0


def is_check(board, color):
    return is_attacked(get_king(board,
color), board, opposing_color(color))


def get_attacks(moving_coin, board, color):
    coin = board[bb2index(moving_coin)]

    if coin & COIN_MASK == PAWN:
        return pawn_attacks(moving_coin,
board, color)
    elif coin & COIN_MASK == KNIGHT:
        return knight_attacks(moving_coin)
    elif coin & COIN_MASK == BISHOP:
        return bishop_attacks(moving_coin,
board, color)
    elif coin & COIN_MASK == ROOK:
        return rook_attacks(moving_coin,
```

```
board, color)
    elif coin & COIN_MASK == QUEEN:
        return queen_attacks(moving_coin,
board, color)
    elif coin & COIN_MASK == KING:
        return king_attacks(moving_coin)


def get_moves(moving_coin, game, color):
    coin = game.board[bb2index(moving_coin)]

    if coin & COIN_MASK == PAWN:
        return pawn_moves(moving_coin, game,
color)
    elif coin & COIN_MASK == KNIGHT:
        return knight_moves(moving_coin,
game.board, color)
    elif coin & COIN_MASK == BISHOP:
        return bishop_moves(moving_coin,
game.board, color)
    elif coin & COIN_MASK == ROOK:
        return rook_moves(moving_coin,
game.board, color)
    elif coin & COIN_MASK == QUEEN:
        return queen_moves(moving_coin,
game.board, color)
    elif coin & COIN_MASK == KING:
        return king_moves(moving_coin,
game.board, color)


def count_attacks(target, board,
attacking_color):
    attack_count = 0

    for index in range(64):
        coin = board[index]
        if coin != EMPTY and coin &
COLOR_MASK == attacking_color:
            pos = 0b1 << index

            if get_attacks(pos, board,
attacking_color) & target:
                attack_count += 1

    return attack_count


def material_sum(board, color):
    material = 0
    for coin in board:
        if coin & COLOR_MASK == color:
            material += COIN_VALUES[coin &
COIN_MASK]
    return material


def material_balance(board):
    return material_sum(board, WHITE) -
material_sum(board, BLACK)
```

```
def mobility_balance(game):
    return count_legal_moves(game, WHITE) -
count_legal_moves(game, BLACK)


def evaluate_game(game):
    if game_ended(game):
        return evaluate_end_node(game)
    else:
        return material_balance(game.board)
+ positional_balance(game)  # +
10*mobility_balance(game)


def evaluate_end_node(game):
    if is_checkmate(game, game.to_move):
        return win_score(game.to_move)
    elif is_stalemate(game) or \
            has_insufficient_material(game)
or \
            is_under_75_move_rule(game):
        return 0


def positional_balance(game):
    return positional_table(game, WHITE) -
positional_table(game, BLACK)


def positional_table(game, color):
    table = 0

    if color == WHITE:
        board = game.board
    elif color == BLACK:
        board = flip_board_v(game.board)

    for index in range(64):
        coin = board[index]

        if coin != EMPTY and coin &
COLOR_MASK == color:
            coin_type = coin & COIN_MASK

            if coin_type == PAWN:
                table += PAWN_TABLE[index]
            elif coin_type == KNIGHT:
                table += KNIGHT_TABLE[index]
            elif coin_type == BISHOP:
                table += BISHOP_TABLE[index]

            elif coin_type == ROOK:
                position = 0b1 << index

                if is_open_file(position,
board):
                    table +=
ROOK_OPEN_FILE_TABLE
                elif
is_semi_open_file(position, board):
```

```
                    table +=
ROOK_SEMI_OPEN_FILE_TABLE

                if position & RANK_7:
                    table +=
ROOK_ON_SEVENTH_TABLE

            elif coin_type == KING:
                if is_endgame(board):
                    table +=
KING_ENDGAME_TABLE[index]
                else:
                    table +=
KING_TABLE[index]

    return table


def is_endgame(board):
    return
count_coins(occupied_squares(board)) <=
ENDGAME_COIN_COUNT


def is_open_file(bitboard, board):
    for f in FILES:
        rank_filter = get_file(f)
        if bitboard & rank_filter:
            return
count_coins(get_all_pawns(board) &
rank_filter) == 0


def is_semi_open_file(bitboard, board):
    for f in FILES:
        rank_filter = get_file(f)
        if bitboard & rank_filter:
            return
count_coins(get_all_pawns(board) &
rank_filter) == 1


def count_coins(bitboard):
    return bin(bitboard).count("1")


def win_score(color):
    if color == WHITE:
        return -10 * COIN_VALUES[KING]
    if color == BLACK:
        return 10 * COIN_VALUES[KING]


def pseudo_legal_moves(game, color):
    for index in range(64):
        coin = game.board[index]

        if coin != EMPTY and coin &
COLOR_MASK == color:
            coin_pos = 0b1 << index
```

```
            for target in
single_gen(get_moves(coin_pos, game,
color)):
                yield (coin_pos, target)

    if can_castle_kingside(game, color):
        yield (get_king(game.board, color),
east_one(east_one(get_king(game.board,
color))))
    if can_castle_queenside(game, color):
        yield (get_king(game.board, color),
west_one(west_one(get_king(game.board,
color))))


def legal_moves(game, color):
    for move in pseudo_legal_moves(game,
color):
        if is_legal_move(game, move):
            yield move


def is_legal_move(game, move):
    new_game = make_move(game, move)
    return not is_check(new_game.board,
game.to_move)


def count_legal_moves(game, color):
    move_count = 0
    for _ in legal_moves(game, color):
        move_count += 1
    return move_count


def is_stalemate(game):
    for _ in legal_moves(game,
game.to_move):
        return False
    return not is_check(game.board,
game.to_move)


def is_checkmate(game, color):
    for _ in legal_moves(game,
game.to_move):
        return False
    return is_check(game.board, color)


def is_same_position(FEN_a, FEN_b):
    FEN_a_list = FEN_a.split(' ')
    FEN_b_list = FEN_b.split(' ')
    return FEN_a_list[0] == FEN_b_list[0]
and \
           FEN_a_list[1] == FEN_b_list[1]
and \
           FEN_a_list[2] == FEN_b_list[2]
and \
           FEN_a_list[3] == FEN_b_list[3]
```

```
def has_threefold_repetition(game):
    current_pos = game.position_history[-1]
    position_count = 0
    for position in game.position_history:
        if is_same_position(current_pos,
position):
            position_count += 1
    return position_count >= 3


def is_under_50_move_rule(game):
    return game.halfmove_clock >= 100


def is_under_75_move_rule(game):
    return game.halfmove_clock >= 150


def has_insufficient_material(game):  #
TODO: other insufficient positions
    if material_sum(game.board, WHITE) +
material_sum(game.board, BLACK) == 2 *
COIN_VALUES[KING]:
        return True
    if material_sum(game.board, WHITE) ==
COIN_VALUES[KING]:
        if material_sum(game.board, BLACK)
== COIN_VALUES[KING] + COIN_VALUES[KNIGHT]
and \
                (get_knights(game.board,
BLACK) != 0 or get_bishops(game.board,
BLACK) != 0):
            return True
    if material_sum(game.board, BLACK) ==
COIN_VALUES[KING]:
        if material_sum(game.board, WHITE)
== COIN_VALUES[KING] + COIN_VALUES[KNIGHT]
and \
                (get_knights(game.board,
WHITE) != 0 or get_bishops(game.board,
WHITE) != 0):
            return True
    return False


def game_ended(game):
    return is_checkmate(game, WHITE) or \
           is_checkmate(game, BLACK) or \
           is_stalemate(game) or \
           has_insufficient_material(game)
or \
           is_under_75_move_rule(game)


def random_move(game, color):
    return choice(legal_moves(game, color))


def evaluated_move(game, color):
    best_score = win_score(color)
    best_moves = []

    for move in legal_moves(game, color):
        evaluation =
evaluate_game(make_move(game, move))

        if is_checkmate(make_move(game,
move), opposing_color(game.to_move)):
            return [move, evaluation]

        if (color == WHITE and evaluation >
best_score) or \
                (color == BLACK and
evaluation < best_score):
            best_score = evaluation
            best_moves = [move]
        elif evaluation == best_score:
            best_moves.append(move)

    return [choice(best_moves), best_score]


def minimax(game, color, depth=1):
    if game_ended(game):
        return [None, evaluate_game(game)]

    [simple_move, simple_evaluation] =
evaluated_move(game, color)

    if depth == 1 or \
            simple_evaluation ==
win_score(opposing_color(color)):
        return [simple_move,
simple_evaluation]

    best_score = win_score(color)
    best_moves = []

    for move in legal_moves(game, color):
        his_game = make_move(game, move)

        if is_checkmate(his_game,
his_game.to_move):
            return [move,
win_score(his_game.to_move)]

        [_, evaluation] = minimax(his_game,
opposing_color(color), depth - 1)

        if evaluation ==
win_score(opposing_color(color)):
            return [move, evaluation]

        if (color == WHITE and evaluation >
best_score) or \
                (color == BLACK and
evaluation < best_score):
            best_score = evaluation
            best_moves = [move]
        elif evaluation == best_score:
            best_moves.append(move)
```

```
    return [choice(best_moves), best_score]


def alpha_beta(game, color, depth, alpha=-
float('inf'), beta=float('inf')):
    if game_ended(game):
        return [None, evaluate_game(game)]

    [simple_move, simple_evaluation] =
evaluated_move(game, color)

    if depth == 1 or \
            simple_evaluation ==
win_score(opposing_color(color)):
        return [simple_move,
simple_evaluation]

    best_moves = []

    if color == WHITE:
        for move in legal_moves(game,
color):
            if verbose:
                print('\t' * depth +
str(depth) + '. evaluating ' + COIN_CODES[
                    get_coin(game.board,
move[0])] + move2str(move))

            new_game = make_move(game, move)
            [_, score] =
alpha_beta(new_game, opposing_color(color),
depth - 1, alpha, beta)

            if verbose:
                print('\t' * depth +
str(depth) + '. ' + str(score) + '
[{},{}]'.format(alpha, beta))

            if score ==
win_score(opposing_color(color)):
                return [move, score]

            if score == alpha:
                best_moves.append(move)
            if score > alpha:  # white
maximizes her score
                alpha = score
                best_moves = [move]
                if alpha > beta:  # alpha-
beta cutoff
                    if verbose:
                        print('\t' * depth +
'cutoff')
                    break
        if best_moves:
            return [choice(best_moves),
alpha]
        else:
            return [None, alpha]
    if color == BLACK:
        for move in legal_moves(game,
color):
            if verbose:
                print('\t' * depth +
str(depth) + '. evaluating ' + COIN_CODES[
                    get_coin(game.board,
move[0])] + move2str(move))

            new_game = make_move(game, move)
            [_, score] =
alpha_beta(new_game, opposing_color(color),
depth - 1, alpha, beta)

            if verbose:
                print('\t' * depth +
str(depth) + '. ' + str(score) + '
[{},{}]'.format(alpha, beta))

            if score ==
win_score(opposing_color(color)):
                return [move, score]

            if score == beta:
                best_moves.append(move)
            if score < beta:  # black
minimizes his score
                beta = score
                best_moves = [move]
                if alpha > beta:  # alpha-
beta cutoff
                    if verbose:
                        print('\t' * depth +
'cutoff')
                    break
        if best_moves:
            return [choice(best_moves),
beta]
        else:
            return [None, beta]


def parse_move_code(game, move_code):
    move_code = move_code.replace(" ", "")
    move_code = move_code.replace("x", "")

    if move_code.upper() == 'O-O' or
move_code == '0-0':
        if can_castle_kingside(game,
game.to_move):
            return
castle_kingside_move(game)

    if move_code.upper() == 'O-O-O' or
move_code == '0-0-0':
        if can_castle_queenside(game,
game.to_move):
            return
castle_queenside_move(game)

    if len(move_code) < 2 or len(move_code)
```

```
> 4:
        return False

    if len(move_code) == 4:
        filter_squares =
get_filter(move_code[1])
    else:
        filter_squares = ALL_SQUARES

    destination_str = move_code[-2:]
    if destination_str[0] in FILES and
destination_str[1] in RANKS:
        target_square =
str2bb(destination_str)
    else:
        return False

    if len(move_code) == 2:
        coin = PAWN
    else:
        coin_code = move_code[0]
        if coin_code in FILES:
            coin = PAWN
            filter_squares =
get_filter(coin_code)
        elif coin_code in COIN_CODES:
            coin = COIN_CODES[coin_code] &
COIN_MASK
        else:
            return False

    valid_moves = []
    for move in legal_moves(game,
game.to_move):
        if move[1] & target_square and \
                move[0] & filter_squares and
\
                get_coin(game.board,
move[0]) & COIN_MASK == coin:
            valid_moves.append(move)

    if len(valid_moves) == 1:
        return valid_moves[0]
    else:
        return False


def get_player_move(game):
    move = None
    while not move:
        move = parse_move_code(game,
input())
        if not move:
            print('Invalid move!')
    return move


def get_AI_move(game, depth=2):
    if verbose:
        print('Searching best move for
white...' if game.to_move == WHITE else
'Searching best move for black...')
    start_time = time()

    if find_in_guide(game):
        move = get_guide_move(game)
    else:
        # move = minimax(game, game.to_move,
depth)[0]
        move = alpha_beta(game,
game.to_move, depth)[0]

    end_time = time()
    if verbose:
        print('Found move ' +
COIN_CODES[get_coin(game.board, move[0])] +
' from ' + str(
            bb2str(move[0])) + ' to ' +
str(bb2str(move[1])) + ' in {:.3f}
seconds'.format(
            end_time - start_time) + '
({},{})'.format(evaluate_game(game),
evaluate_game(make_move(game, move))))
    return move


def print_outcome(game):
    print(get_outcome(game))


def get_outcome(game):
    if is_stalemate(game):
        return 'Draw by stalemate'
    if is_checkmate(game, WHITE):
        return 'BLACK wins!'
    if is_checkmate(game, BLACK):
        return 'WHITE wins!'
    if has_insufficient_material(game):
        return 'Draw by insufficient
material!'
    if is_under_75_move_rule(game):
        return 'Draw by 75-move rule!'


def play_as_white(game=Game()):
    print('Playing as white!')
    while True:
        print_board(game.board)
        if game_ended(game):
            break

        game = make_move(game,
get_player_move(game))

        print_board(game.board)
        if game_ended(game):
            break

        game = make_move(game,
get_AI_move(game))
    print_outcome(game)
```

```
def play_as_black(game=Game()):
    print('Playing as black!')
    while True:
        print_rotated_board(game.board)
        if game_ended(game):
            break

        game = make_move(game,
get_AI_move(game))

        print_rotated_board(game.board)
        if game_ended(game):
            break

        game = make_move(game,
get_player_move(game))
    print_outcome(game)


def watch_AI_game(game=Game(),
sleep_seconds=0):
    print('Watching AI-vs-AI game!')
    while True:
        print_board(game.board)
        if game_ended(game):
            break

        game = make_move(game,
get_AI_move(game))
        sleep(sleep_seconds)
    print_outcome(game)


def play_as(color):
    if color == WHITE:
        play_as_white()
    if color == BLACK:
        play_as_black()


def play_random_color():
    color = choice([WHITE, BLACK])
    play_as(color)


def find_in_guide(game):
    if game.position_history[0] !=
INITIAL_FEN:
        return False

    openings = []
    guide_file = open("guide.txt")
    for line in guide_file:
        if
line.startswith(game.get_move_list()) and
line.rstrip() > game.get_move_list():
            openings.append(line.rstrip())
    guide_file.close()
    return openings
```

```
def get_guide_move(game):
    openings = find_in_guide(game)
    chosen_opening = choice(openings)
    next_moves =
chosen_opening.replace(game.get_move_list(),
'').lstrip()
    move_str = next_moves.split(' ')[0]
    move = [str2bb(move_str[:2]),
str2bb(move_str[-2:])]
    return move
```

The below code is Main.py

```
import pygame, Engine
from random import choice
from traceback import format_exc
from sys import stderr
from time import strftime
from copy import deepcopy

pygame.init()

SQ_SIDE = 50
AI_SEARCH_DEPTH = 2

RED_CHECK = (240, 150, 150)  # when check sq
becomes red
GRAY_LIGHT = (240, 240, 240)
GRAY_DARK = (200, 200, 200)
CHESSWEBSITE_LIGHT = (212, 202, 190)
CHESSWEBSITE_DARK = (100, 92, 89)

BOARD_COLORS = [(GRAY_LIGHT, GRAY_DARK),
(CHESSWEBSITE_LIGHT, CHESSWEBSITE_DARK)]
BOARD_COLOR = choice(BOARD_COLORS)

BK = pygame.image.load('Coins/BK.png')
BQ = pygame.image.load('Coins/BQ.png')
BR = pygame.image.load('Coins/BR.png')
BB = pygame.image.load('Coins/BB.png')
BN = pygame.image.load('Coins/BN.png')
BP = pygame.image.load('Coins/BP.png')

WK = pygame.image.load('Coins/WK.png')
WQ = pygame.image.load('Coins/WQ.png')
WR = pygame.image.load('Coins/WR.png')
WB = pygame.image.load('Coins/WB.png')
WN = pygame.image.load('Coins/WN.png')
WP = pygame.image.load('Coins/WP.png')

CLOCK = pygame.time.Clock()
CLOCK_TICK = 10

SCREEN = pygame.display.set_mode((8 *
SQ_SIDE, 8 * SQ_SIDE), pygame.RESIZABLE)
SCREEN_TITLE = 'Chess Game'

pygame.display.set_icon(pygame.image.load('C
oins/Chess logo.ico'))
pygame.display.set_caption(SCREEN_TITLE)
```

```
def resize_screen(sq_side_len):
    global SQ_SIDE
    global SCREEN
    SCREEN = pygame.display.set_mode((8 *
sq_side_len, 8 * sq_side_len),
pygame.RESIZABLE)
    SQ_SIDE = sq_side_len


def print_empty_board():
    SCREEN.fill(BOARD_COLOR[0])
    paint_dark_squares(BOARD_COLOR[1])


def paint_sq(sq, sq_color):
    col = Engine.FILES.index(sq[0])
    row = 7 - Engine.RANKS.index(sq[1])
    pygame.draw.rect(SCREEN, sq_color,
(SQ_SIDE * col, SQ_SIDE * row, SQ_SIDE,
SQ_SIDE), 0)


def paint_dark_squares(sq_color):
    for position in
Engine.single_gen(Engine.DARK_SQUARES):
        paint_sq(Engine.bb2str(position),
sq_color)


def get_sq_rect(sq):
    col = Engine.FILES.index(sq[0])
    row = 7 - Engine.RANKS.index(sq[1])
    return pygame.Rect((col * SQ_SIDE, row *
SQ_SIDE), (SQ_SIDE, SQ_SIDE))


def coord2str(position, color=Engine.WHITE):
    if color == Engine.WHITE:
        file_index = int(position[0] /
SQ_SIDE)
        rank_index = 7 - int(position[1] /
SQ_SIDE)
        return Engine.FILES[file_index] +
Engine.RANKS[rank_index]
    if color == Engine.BLACK:
        file_index = 7 - int(position[0] /
SQ_SIDE)
        rank_index = int(position[1] /
SQ_SIDE)
        return Engine.FILES[file_index] +
Engine.RANKS[rank_index]


def print_board(board, color=Engine.WHITE):
    if color == Engine.WHITE:
        printed_board = board
    if color == Engine.BLACK:
        printed_board =
Engine.rotate_board(board)
    print_empty_board()

    if Engine.is_check(board, Engine.WHITE):

paint_sq(Engine.bb2str(Engine.get_king(print
ed_board, Engine.WHITE)), RED_CHECK)
    if Engine.is_check(board, Engine.BLACK):

paint_sq(Engine.bb2str(Engine.get_king(print
ed_board, Engine.BLACK)), RED_CHECK)

    for position in
Engine.colored_coin_gen(printed_board,
Engine.KING, Engine.BLACK):

SCREEN.blit(pygame.transform.scale(BK,
(SQ_SIDE, SQ_SIDE)),

get_sq_rect(Engine.bb2str(position)))
    for position in
Engine.colored_coin_gen(printed_board,
Engine.QUEEN, Engine.BLACK):

SCREEN.blit(pygame.transform.scale(BQ,
(SQ_SIDE, SQ_SIDE)),

get_sq_rect(Engine.bb2str(position)))
    for position in
Engine.colored_coin_gen(printed_board,
Engine.ROOK, Engine.BLACK):

SCREEN.blit(pygame.transform.scale(BR,
(SQ_SIDE, SQ_SIDE)),

get_sq_rect(Engine.bb2str(position)))
    for position in
Engine.colored_coin_gen(printed_board,
Engine.BISHOP, Engine.BLACK):

SCREEN.blit(pygame.transform.scale(BB,
(SQ_SIDE, SQ_SIDE)),

get_sq_rect(Engine.bb2str(position)))
    for position in
Engine.colored_coin_gen(printed_board,
Engine.KNIGHT, Engine.BLACK):

SCREEN.blit(pygame.transform.scale(BN,
(SQ_SIDE, SQ_SIDE)),

get_sq_rect(Engine.bb2str(position)))
    for position in
Engine.colored_coin_gen(printed_board,
Engine.PAWN, Engine.BLACK):

SCREEN.blit(pygame.transform.scale(BP,
(SQ_SIDE, SQ_SIDE)),

get_sq_rect(Engine.bb2str(position)))
```

```
    for position in
Engine.colored_coin_gen(printed_board,
Engine.KING, Engine.WHITE):

SCREEN.blit(pygame.transform.scale(WK,
(SQ_SIDE, SQ_SIDE)),

get_sq_rect(Engine.bb2str(position)))
    for position in
Engine.colored_coin_gen(printed_board,
Engine.QUEEN, Engine.WHITE):

SCREEN.blit(pygame.transform.scale(WQ,
(SQ_SIDE, SQ_SIDE)),

get_sq_rect(Engine.bb2str(position)))
    for position in
Engine.colored_coin_gen(printed_board,
Engine.ROOK, Engine.WHITE):

SCREEN.blit(pygame.transform.scale(WR,
(SQ_SIDE, SQ_SIDE)),

get_sq_rect(Engine.bb2str(position)))
    for position in
Engine.colored_coin_gen(printed_board,
Engine.BISHOP, Engine.WHITE):

SCREEN.blit(pygame.transform.scale(WB,
(SQ_SIDE, SQ_SIDE)),

get_sq_rect(Engine.bb2str(position)))
    for position in
Engine.colored_coin_gen(printed_board,
Engine.KNIGHT, Engine.WHITE):

SCREEN.blit(pygame.transform.scale(WN,
(SQ_SIDE, SQ_SIDE)),

get_sq_rect(Engine.bb2str(position)))
    for position in
Engine.colored_coin_gen(printed_board,
Engine.PAWN, Engine.WHITE):

SCREEN.blit(pygame.transform.scale(WP,
(SQ_SIDE, SQ_SIDE)),

get_sq_rect(Engine.bb2str(position)))

    pygame.display.flip()


def set_title(title):
    pygame.display.set_caption(title)
    pygame.display.flip()


def make_AI_move(game, color):
    set_title(SCREEN_TITLE + " - Machine
Playing...")
    new_game = Engine.make_move(game,
Engine.get_AI_move(game, AI_SEARCH_DEPTH))
    set_title(SCREEN_TITLE)
    print_board(new_game.board, color)
    return new_game


def try_move(game, attempted_move):
    for move in Engine.legal_moves(game,
game.to_move):
        if move == attempted_move:
            game = Engine.make_move(game,
move)
    return game


def play_as(game, color):
    run = True
    ongoing = True

    try:
        while run:
            CLOCK.tick(CLOCK_TICK)
            print_board(game.board, color)

            if Engine.game_ended(game):
                set_title(SCREEN_TITLE + ' -
' + Engine.get_outcome(game))
                ongoing = False

            if ongoing and game.to_move ==
Engine.opposing_color(color):
                game = make_AI_move(game,
color)

            if Engine.game_ended(game):
                set_title(SCREEN_TITLE + ' -
' + Engine.get_outcome(game))
                ongoing = False

            for event in pygame.event.get():
                if event.type ==
pygame.QUIT:
                    run = False

                if event.type ==
pygame.MOUSEBUTTONDOWN:
                    leaving_sq =
coord2str(event.pos, color)

                if event.type ==
pygame.MOUSEBUTTONUP:
                    arriving_sq =
coord2str(event.pos, color)

                    if ongoing and
game.to_move == color:
                        move =
(Engine.str2bb(leaving_sq),
Engine.str2bb(arriving_sq))
                        game =
try_move(game, move)
```

```
print_board(game.board, color)

                if event.type ==
pygame.KEYDOWN:
                    if event.key ==
pygame.K_ESCAPE or event.key == 113:
                        run = False
                    if event.key == 104 and
ongoing:  # H key
                        game =
make_AI_move(game, color)
                    if event.key == 117:  #
U key
                        game =
Engine.unmake_move(game)
                        game =
Engine.unmake_move(game)

set_title(SCREEN_TITLE)

print_board(game.board, color)
                        ongoing = True
                    if event.key == 99:  # C
key
                        global BOARD_COLOR
                        new_colors =
deepcopy(BOARD_COLORS)

new_colors.remove(BOARD_COLOR)
                        BOARD_COLOR =
choice(new_colors)

print_board(game.board, color)
                    if event.key == 112 or
event.key == 100:  # P or D key

print(game.get_move_list() + '\n')

print('\n'.join(game.position_history))
                    if event.key == 101:  #
E key
                        print('eval = ' +
str(Engine.evaluate_game(game) / 100))

                if event.type ==
pygame.VIDEORESIZE:
                    if SCREEN.get_height()
!= event.h:
```

```
resize_screen(int(event.h / 8.0))
                    elif SCREEN.get_width()
!= event.w:

resize_screen(int(event.w / 8.0))
                    print_board(game.board,
color)
    except:
        print(format_exc(), file=stderr)
        bug_file = open('bug_report.txt',
'a')
        bug_file.write('----- ' +
strftime('%x %X') + ' -----\n')
        bug_file.write(format_exc())
        bug_file.write('\nPlaying as
WHITE:\n\t' if color == Engine.WHITE else
'\nPlaying as BLACK:\n\t')
        bug_file.write(game.get_move_list()
+ '\n\t')

bug_file.write('\n\t'.join(game.position_his
tory))
        bug_file.write('\n------------------
-----------\n\n')
        bug_file.close()


def play_as_black(game=Engine.Game()):
    return play_as(game, Engine.BLACK)


def play_as_white(game=Engine.Game()):
    return play_as(game, Engine.WHITE)


def play_random_color(game=Engine.Game()):
# This is to select random color to start
the game
    color = choice([Engine.WHITE,
Engine.BLACK])
    play_as(game, color)


play_random_color()
```